インテル® インストルメント & トレース・テクノロジー（ITT）およびジャストインタイム（JIT）API

このドキュメントは、インテルの GitHub で公開されている 2025 年 11 月 10 日時点の「The Intel® Instrumentation and Tracing Technology (ITT) and Just-In-Time (JIT) APIs」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

本ドキュメントはレイアウト調整および校閲を行っておりません。誤字脱字、製品名や用語の表記、レイアウト等の不具合が含まれる可能性があることを予めご了承ください。

インテル® インストルメント & トレース・テクノロジー（ITT）およびジャストインタイム（JIT）API は、インテルのソフトウェア・ツールと組み合わせて使用するオープンソースのプロファイル API であり、パフォーマンス解析中のトレースデータ収集および管理に使用されます。インテル® VTune™ プロファイラーおよびインテル® グラフィックス・パフォーマンス・アナライザー（インテル® GPA）を使用する際に、ITT/JIT API を使用してプロファイルを行うことができます。

このリポジトリーには、これらの API の使用方法を説明するドキュメントが含まれています。

内容:

概要

ソフトウェア・アプリケーションのパフォーマンス向上にインテルのアナライザー・ツールを使用する場合、実行時に、インテル® インストルメントとトレース・テクノロジー（ITT）およびジャストインタイム（JIT）API を使用してコードを インストルメントし、トレースデータを生成してその収集を制御します。ITT/JIT API を使用してコード内の特定の部分を測定することで、パフォーマンスのボトルネックやリソース使用状況に関する情報を得ることができます。

コンポーネント

- **ITT API**: アプリケーションの実行中にトレースデータの生成と収集を制御する機能を強化し、インテルツールとのシームレスな統合を実現します。
- **JIT API**: このレポートには、ジャストインタイム（JIT）コンパイルされたコードに関する詳細情報が含まれており、動的に生成されたコードのパフォーマンスをプロファイルできます。

アーキテクチャー

ITT/JIT API は 2 つの部分で構成されています:

- **静的パート**: トレース機能を有効にするため、アプリケーションにコンパイルしてリンクするオープンソースの静的ライブラリー（ittapi）。
- **動的パート**: 特定のツール向けに、トレースデータを収集して書き込む共有ライブラリー。動的パートのリファレンス実装については、リファレンス・コレクターを確認してください。

ソースコードからビルド

技術要件

ITT/JIT API をビルドする前に、以下のハードウェアおよびソフトウェア・ツールが揃っていることを確認してください:

- C/C++ コンパイラーを含む、一般的な開発ツールを入手してください
- Python 3.6 以降をインストール
- CMake 3.5 以降をインストール
- Windows* システムでは、以下のいずれかをインストールしてください:
  - Microsoft* Visual Studio 2015 以降
  - Ninja 1.9 以降
- Fortran のサポートを有効にするには、インテル® Fortran コンパイラーをインストールしてください

ITT/JIT API のソースコードを入手

ITT/JIT API のソースコードを入手するには以下の方法があります:

- 最新の公開リリース版からダウンロード

- リポジトリーのクローンを作成します:

```
git clone https://github.com/intel/ittapi.git
```

ITT / JIT API をビルド

ITT/JIT API の静的ライブラリーをビルドするには、以下のコマンドを実行します:

```
python buildall.py <options>
```

これらのオプションを使用して、ビルドプロセスを設定します:

```
usage: python buildall.py [-h] [-d] [-c] [-v] [-pt] [-ft] [--force_bits]

optional arguments:
  -h, --help      show this help message and exit
  -d, --debug     specify debug build configuration (release by default)
  -c, --clean     delete any intermediate and output files
  -v, --verbose   enable verbose output from build process
  -pt, --ptmark   enable anomaly detection support
  -ft, --fortran  enable fortran support
  --force_bits    specify bit version for the target
  --vs           specify visual studio version (Windows only)
  --cmake_gen     specify cmake build generator (Windows only)
```

ITT / JIT API を使用

このセクションでは、ITT/JIT API を様々な環境で使用する方法について説明します。ITT/JIT API は一連の C/C++ 関数で構成されており、Java* や .NET* コードは使用していません。ランタイム環境に関するサポートが必要な場合は、マネージドコードから Java ネイティブ・インターフェイス（JNI）または C/C++ 関数呼び出しを使用してください。

C/C++ API の使用方法とリファレンス:

- インストルメントとトレース・テクノロジー（ITT）API
- ジャストインタイム（JIT）API

その他の言語 API とバインド:

- Rust ITT API とバインド（英語）
- Python ITT API とバインド(英語)

インストルメントとトレース・テクノロジー（ITT）API

インテル® インストルメントとトレース・テクノロジー（ITT）API を使用して、アプリケーションの実行中にトレースデータを生成して収集を制御できます。

ITT API を使用して以下を行います:

- 収集したトレース量に基づいてアプリケーションのパフォーマンス・オーバーヘッドを制御します。
- アプリケーションを再コンパイルすることなくトレース収集を有効にします
- さらに詳細な解析にはコード・アノテーションを有効にします。

ITT API を使用すると、Windows*、Linux*、または FreeBSD* システム上で動作する C、C++、または Fortran アプリケーションのレースデータを収集できます。

ITT API は、静的ライブラリー・コンポーネントと動的ライブラリー・コンポーネントで構成されています。静的ライブラリーにリンクするアプリケーションやモジュールは、動的ライブラリーに対する実行時依存関係を持ちません。したがって、これらのコンポーネントは独立して実行できます。

ITT API の使用方法とリファレンス

- ITT API を使用したコンパイルとリンク
- アプリケーションをインストルメントする
- ITT API のオーバーヘッドを最小化する
- ITT API リファレンス

ITT API を使用したコンパイルとリンク

ステップ 1: ビルドシステムの設定

ITT API を使用してアプリケーションにインストルメント機能を追加する前に、ビルドシステムを設定して API のヘッダーファイルとライブラリーにアクセスできるようにします:

- `INCLUDE` パスに `<ittapi_dir>¥include` を追加します
- `LIBRARIES` パスに `<ittapi_dir>¥build_<target_platform>¥<target_bits>¥bin` を追加します

ステップ 2: アプリケーションで ITT API ヘッダー/モジュールをインクルード

C/C++ アプリケーション

インストルメント対象のすべてのソースファイルに、以下の `#include` 文を追加します:

```
#include <ittnotify.h>
```

`ittnotify.h` ヘッダーには、ITT API ルーチンと、アプリケーションから API を適切に呼び出すロジックを提供するマクロ定義が含まれています。

トレースが無効になっている場合、ITT API によるオーバーヘッドはほぼゼロです。オーバーヘッドを完全にゼロにするには、アプリケーションのすべての ITT API 呼び出しをコンパイルしないようにできます。これを行うに

は、 `ittnotify.h` ファイルをインクルードする前に、コンパイル時にプロジェクト内で `INTEL_NO_ITTNOTIFY_API` マクロを定義します。これはコンパイラーのコマンドラインから行うことも、ソースファイル内で行うこともできます。

Fortran アプリケーション

ソースファイルに `ITTNOTIFY` モジュールを追加します。以下のソース行を使用します:

```
USE ITTNOTIFY
```

ステップ 3: アプリケーションに ITT 通知を挿入

以下を使用して、アプリケーションに ITT 通知を挿入:

言語	通知	例
C/C++	`__itt_*`	`__itt_pause();`
Fortran	`ITT_*`	`CALL ITT_PAUSE()`

さらに詳しく:

- アプリケーションをインストルメントする
- ITT API リファレンス

ステップ 4: libittnotify 静的ライブラリーをアプリケーションにリンクします

アプリケーションへの ITT 通知の挿入が完了したら、次のステップで静的ライブラリー libittnotify をリンクします。このライブラリーは、Linux* および FreeBSD* システムでは `libittnotify.a`、Windows* システムでは `libittnotify.lib` です。

トレースを有効にしている場合、静的ライブラリーは ITT API データの動的コレクターをロードし、ITT API からの計測データをコレクターに転送します。

トレースが無効にされていると、静的ライブラリーは ITT API 呼び出しを無視するため、インストルメントのオーバーヘッドはほぼゼロになります。

ステップ 5: 動的ライブラリーをロード

アプリケーションにインストルメント・コードを組み込み、静的ライブラリーをリンクした後、ITT API の動的ライブラリーをアプリケーションにロードする必要があります。これには、システム・アーキテクチャーに応じて、`INTEL_LIBITTNOTIFY32` または `INTEL_LIBITTNOTIFY64` 環境変数を設定します。

Windows*:

```
set INTEL_LIBITTNOTIFY32=<install-dir>¥bin32¥runtime¥ittnotify_collector.dll
set INTEL_LIBITTNOTIFY64=<install-dir>¥bin64¥runtime¥ittnotify_collector.dll
```

Linux*:

```
export INTEL_LIBITTNOTIFY32=<install-dir>/lib32/runtime/libittnotify_collector.so
export INTEL_LIBITTNOTIFY64=<install-dir>/lib64/runtime/libittnotify_collector.so
```

FreeBSD*:

```
setenv INTEL_LIBITTNOTIFY64=<target-
package>/lib64/runtime/libittnotify_collector.so
```

参考情報: Unicode のサポート

`__itt_char` パラメーターを持つすべての API 関数は、Windows* の Unicode 規則に従います。

Windows* システム上でコンパイルされると、`UNICODE` マクロが定義されている場合、`__itt_char` は `wchar_t` に設定されます。`UNICODE` マクロが定義されていない場合、`__itt_char` は `char` に設定されます。

実際の関数名は、ASCII API では `A`、unicode API では `W` サフィックスが付きます。どちらも関数は API を実装する DLL で定義されています。

ASCII キャラクターのみの文字列は、Unicode と ASCII API バージョンの両方で内部的に等価です。例えば、以下の文字列は等価です:

```
__itt_sync_createA( addr, "OpenMP Scheduler", "Critical Section", 0);
__itt_sync_createW( addr, L"OpenMP Scheduler", L"Critical Section", 0);
```

アプリケーションをインストルメントする

ITT/JIT API を使用してパフォーマンス・データを収集する場合、最適な結果を得るためコード内に API 呼び出しを追加して論理的なタスクを指定します。これは、ほかの CPU と GPU タスクに関連するコード中のタスク間の関係（開始と終了を含む）を可視化するのに役立ちます。

最も基本的なレベルでは、タスクとは、特定の単一スレッド上で実行されるワークの論理的なグループのことです。タスクとは、プログラム内のコードで開発者が重要だと考えるグループに対応させることができます。`__itt_task_begin` と `__itt_task_end` 呼び出しを使用して、論理タスクの始まりと終わりを識別するするためコードをマークアップします。

開始するには、次の API 呼び出しを使用します:

- `__itt_domain_create()` ITT API 呼び出しを識別するドメインを作成します。少なくとも 1 つのドメインを定義します。
- `__itt_string_handle_create()` タスクを識別する文字列ハンドルを作成します。トレースを識別するには、文字列よりも文字列ハンドルの方が効率的です。
- `__itt_task_begin()` タスクの開始をマークします。
- `__itt_task_end()` タスクの終了をマークします。

例

この例は、マルチスレッド・アプリケーションで 4 つの基本的な ITT API 関数の使い方を示しています:

- ドメイン API
- 文字列ハンドル API
- タスク API
- スレッド命名 API

```
#include <windows.h>
#include <ittnotify.h>

// スレッド関数の前方宣言。
DWORD WINAPI workerthread(LPVOID);
bool g_done = false;

// グローバルに参照可能なドメインを作成：この例ではこれを使用します。
__itt_domain* domain = __itt_domain_create("Example.Domain.Global");
// "main" タスクに関連付ける文字列ハンドルを作成。
__itt_string_handle* handle_main = __itt_string_handle_create("main");
__itt_string_handle* handle_createthread =
__itt_string_handle_create("CreateThread");

void main(int, char* argv[])
{
  // "main" ルーチンに関連付けられたタスクを作成します。
  __itt_task_begin(domain, __itt_null, __itt_null, handle_main);

  // ここで 4 つのワーカースレッドを作成します
  for (int i = 0; i < 4; i++)
  {
    // CreateThread のコストに注目します。測定のためトレース API を追加します。
    __itt_task_begin(domain, __itt_null, __itt_null, handle_createthread);
    cppCreateThread(NULL, 0, workerthread, (LPVOID)i, 0, NULL);
    __itt_task_end(domain);
```

```
  }

  // while を待機 ...
  cppSleep(5000);
  g_done = true;

  // Main タスクの終了をマーク
  __itt_task_end(domain);
}

// ワークタスク用の文字列ハンドルを作成します。
__itt_string_handle* handle_work = __itt_string_handle_create("work");
DWORD WINAPI workerthread(LPVOID data)
{
  // このスレッドの名前を設定して、UI 上で分かりやすい名前で表示されるようにします
  char threadname[32];
  wsprintf(threadname, "Worker Thread %d", data);
  __itt_thread_set_name(threadname);

  // 各ワーカースレッドは、一定数の "work" タスクを実行します。
  while(!g_done)
  {
    __itt_task_begin(domain, __itt_null, __itt_null, handle_work);
    cppSleep(150);
    __itt_task_end(domain);
  }

  return 0;
}
```

ITT API のオーバーヘッドを最小化する

アプリケーションに追加するインストルメントの範囲によって、ITT API によって発生するオーバーヘッドの量と、アプリケーションのパフォーマンスに与える影響が異なります。このオーバーヘッドを最小化するには、求められるアプリケーション・パフォーマンスと収集するパフォーマンス・データ量のバランスを取ることが重要です。

次のガイドラインに従ってください:

- 解析にとって重要なアプリケーション内のパスのみにインストルメント関数を追加してください。
- ITT ドメインと文字列ハンドルをクリティカル・パス外で作成します。
- 個別に解析できるアプリケーションの異なる側面のデータ収集をフィルターします。無効にされた API 呼び出しのオーバーヘッド（関連する呼び出しをフィルター処理）は、常に 10 クロックティック未満で

す。

条件付きコンパイル

リリースバージョンのコードでは、条件付きコンパイルを使用してアノテーションを無効にします。コンパイル時に `ittnotify.h` をインクルードする前に、`INTEL_NO_ITTNOTIFY_API` マクロを定義して、コードからすべての `__itt_*` 関数を除外します。

このマクロを定義することで、静的ライブラリーをリンクステージから除外することもできます。

使用例:

ITT API にはドメインと文字列ハンドルを作成する関数のサブセットが含まれています。これらの関数は、常にドメイン名と文字列に対して同じハンドルを返します。この処理には、文字列比較とテーブル検索を行う一連の関数が必要です。これらの比較や検索は、深刻なパフォーマンス低下を招く可能性があります。さらに、これらの関数のパフォーマンスは、作成されるドメインや文字列ハンドル数の対数に比例します。良い方法としては、アプリケーションの起動時やグローバルスコープで、ドメインと文字列ハンドルを作成することです。

次のコードセクションでは、グルーバルスコープに 2 つのドメインを作成します。これらのドメインを使用して、トレースファイルに書き込む詳細レベルを制御できます。

```
#include "ittnotify.h"

// グローバルスコープでドメインを作成
__itt_domain* basic = __itt_domain_create(L"MyFunction.Basic");
__itt_domain* detailed = __itt_domain_create(L"MyFunction.Detailed");

// グローバルスコープで文字列ハンドルを作成
__itt_string_handle* h_my_function = __itt_string_handle_create(L"MyFunction");
void MyFunction(int arg)
{
    __itt_task_begin(basic, __itt_null, __itt_null, h_my_function);
    Foo(arg);
    FooEx();
    __itt_task_end(basic);
}

__itt_string_handle* h_foo = __itt_string_handle_create(L"Foo");
void Foo(int arg)
{
    // 詳細ドメインが無効になっている場合は、詳細なデータ追跡をスキップ
    __itt_task_begin(detailed, __itt_null, __itt_null, h_foo);
```

```
    // ここにワークを記述 ...
    __itt_task_end(detailed);
}

__itt_string_handle* h_foo_ex = __itt_string_handle_create(L"FooEx");
void FooEx()
{
    // 詳細ドメインが無効になっている場合は、詳細なデータ追跡をスキップ
    __itt_task_begin(detailed, __itt_null, __itt_null, h_foo_ex);
    // ここにワークを記述 ...
    __itt_task_end(detailed);
}

// ここが出発点
int main(int argc, char** argv)
{
  if(argc < 2)
  {
    // アプリケーションの実行でドメインからのトレースが必要ない場合は、
    // 詳細ドメインを無効にします
    detailed ->flags = 0;
  }

  MyFunction(atoi(argv[1]));
  return 0;
}
```

ITT API リファレンス

- クロックドメイン API
- 収集コントロール API
- コンテキスト・メタデータ API
- カウンター API
- ドメイン API
- イベント API
- 形式指定メタデータ API
- フレーム API
- ヒストグラム API
- ロードモジュール API
- マーカー API
- メモリー割り当て API
- メタデータ API

- リレーション API
- 文字列ハンドル API
- タスク API
- スレッド命名 API
- ユーザー定義の同期 API

クロックドメイン API

一部のアプリケーションでは、CPU によって生成されるタイムスタンプや周波数とは異なる、ユーザー定義のタイムスタンプと周波数でイベントを追跡する機能が必要となることがあります。例えば、GPU 上で発生するイベントをインストルメントする場合があります。これには、クロックドメインを作成できます。

クロックドメインを作成するには、次のプリミティブを使用します:

```
__itt_clock_domain * ITTAPI __itt_clock_domain_create(__itt_get_clock_info_fn fn,
void* fn_data)
```

プリミティブのパラメーター:

[in]	`fn`	代替の CPU タイムスタンプと周波数を取得し、それらをクロックドメイン構造体の `__itt_clock_info` に格納するコールバック関数へのポインター。
[in]	`fn_data`	コールバック関数に渡される引数。`NULL` にできます。

異なるクロックドメインから発行されたタスクが、同じタイムライン上に表示されます。これは、参照されるクロックドメインのベース・タイムスタンプ（クロックドメインが作成された時に取得されたもの）と CPU タイムスタンプ（同じ時に取得されたもの）同期させることで実現されます。

必要に応じて（GPU 周波数が変更された場合などに）、クロックドメインの基準タイムスタンプと周波数を再計算するには以下のプリミティブを使用します:

```
__itt_clock_domain_reset()
```

収集コントロール API

コード内でコレクション制御 API を使用することで、インテル® VTune™ プロファイラーがアプリケーションのデータを収集する方法とタイミングを管理できます。これらの API を呼び出すことで、データ収集を一時停止、再開、または停止することができ、特定のコード領域に解析を集中したり、プロファイルのオーバーヘッドを削減したり、重要でないセクションをパフォーマンス・データの結果から除外したりすることができます。

使用するプリミティブ	説明
`void __itt_pause(void)`	データを収集せずにアプリケーションを実行します。インテル® VTune™ プロファイラーは、スレッドやプロセスの生成など重要な情報のみを収集することで、収集時のオーバーヘッドを軽減できます。
`void __itt_resume(void)`	データの収集を再開します。
`void __itt_detach(void)`	データ収集をデタッチします。インテル® VTune™ プロファイラーはすべてのプロセスからコレクターをデタッチします。アプリケーションはそのまま動作しますが、実行中にデータは収集されません。

データ収集をポーズ

いずれかのスレッドでデータ収集を一時停止すると、アクティブなスレッドだけでなく、プログラム全体のデータ収集が一時停止されます。また、データ収集を一時停止することで、ランタイム解析のオーバーヘッドを削減できます。

影響を受けない API:

- ドメイン API
- 文字列操作 API
- スレッド命名 API

影響を受ける API (一時停止状態ではデータ収集は行われません):

- タスク API
- フレーム API
- イベント API
- ユーザー定義同期 API

注

ポーズ (Pause)/再開 (Resume) API の呼び出し頻度は 1Hz が妥当なレートです。この操作は、解析の実行中にすべてのプロセスのデータ収集を一時停止および再開し、対応する収集の状態を この GUI に通知するため、小さな負荷のワークフローで頻繁に呼び出すことは推奨されません。代わりにフレーム API を使用します。

使用例: 特定のコード領域に注目

このコード例では、一時停止/再開の呼び出しによって、コードの特定の部分のデータ収集に焦点を当てることができます。アプリケーションの実行は、コレクションが一時停止されたときに開始されます。

```
int main(int argc, char* argv[])
{
```

```
  // ここに初期化ワークを記述
  __itt_resume();
    // ここにプロファイル・ワークを記述
  __itt_pause();
  // ここにファイナライズ・ワークを記述
  return 0;
}
```

使用例: コード領域を隠す

この例は、一時的に必要ない集中的な処理を隠すため、一時停止/再開呼び出しをどのように使用するか示しています。

```
int main(int argc, char* argv[])
{
  // ここにワークを記述
  __itt_pause();
  // ここに注目しないワークを記述
  __itt_resume();
  // ここにワークを記述
  __itt_detach();
  // ここに注目しないワークを記述
  return 0;
}
```

コンテキスト・メタデータ API

コンテキスト・メタデータ API を使用すると、特殊属性を使用してコード内にカスタムカウンターを定義できます。インテル® VTune™ プロファイラーでは、収集されたデータのメトリックセットを、帯域幅、レイテンシー、利用率など従来のデータ表現形式で取得することもできます。

コンテキスト・メタデータ API を使用してカウンターベースのメトリックを収集し、次のようなハードウェア・トポロジーに関連付けます:

- PCIe デバイス
- ブロックデバイス
- CPU コア
- スレッド

カウンター・オブジェクトを定義および作成

コンテキスト・メタデータを保存するには、次の構造体を使用します:

```
__itt_context_metadata
{
    __itt_context_type type;     /*!< コンテキスト・メタデータ値のタイプ */
    void* value;                 /*!< コンテキスト・メタデータ値へのポインター */
}
```

この構造体は、次のタイプのコンテキスト・メタデータを受け入れます。

_itt_context_type	値	説明
__itt_context_name	ASCII 文字列 char* / Unicode 文字列 wchar_t*	カウンターベースのメトリックの名前。これは必須値です。
__itt_context_device	ASCII 文字列 char* / Unicode 文字列 wchar_t*	カウンターサンプルを分類する統計サブドメイン（ネットワーク・ポート ID、ディスク・パーティションなど）
__itt_context_units	ASCII 文字列 char* / Unicode 文字列 wchar_t*	測定単位。時間の測定には、ns/us/ms/s 単位を使用して、インテル® VTune™ プロファイラーのデータ表現を変更します。
__itt_context_pci_addr	ASCII 文字列 char* / Unicode 文字列 wchar_t*	カウンターに関連付けるデバイスの PCI アドレス。
__itt_context_tid	符号なし 64 ビット整数タイプ	カウンターに関連付けるスレッド ID。
__itt_context_bandwidth_flag	符号なし 64 ビット整数タイプ（0、1）	このフラグが 1 に設定されている場合、レイテンシー分布図とカウンター/秒のタイムライン分布を計算します。
__itt_context_latency_flag	符号なし 64 ビット整数タイプ（0、1）	このフラグが 1 に設定されている場合、スループット分布図とカウンター/秒のタイムライン分布を計算します。
__itt_context_on_thread_flag	符号なし 64 ビット整数タイプ（0、1）	このフラグが 1 に設定されている場合、スレッドグラフの上部に CPU 時間分布のパーセンテージを示すカウンターが表示されます。

コンテキスト・メタデータをカウンターに関連付ける前に、ITT API ドメインと ITT API カウンター・インスタンスを作成する必要があります。

ドメイン名は、インテル® VTune™ プロファイラーの結果にあるカウンターのメトリック・セクションの見出しを提供します。単一のドメインで、任意の数のカウンターのデータを組み合わせることができます。ただし、カウンターの名前は同一ドメインで同じである必要があります。

コンテキスト・メタデータの単一のメトリックで、さまざまなカウンターを組み合わせることができます。

コンテキスト情報を追加

すべてのオブジェクトを作成したら、選択したカウンターのコンテキスト情報を追加できます。使用するプリミティブ:

```
__itt_bind_context_metadata_to_counter(
    __itt_counter counter, size_t length, __itt_context_metadata* metadata);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`__itt_counter counter`	コンテキスト・メタデータに関連付けられたカウンター・インスタンスへのポインター
[in]	`size_t length`	コンテキスト・メタデータの配列内の要素数
[in]	`__itt_context_metadata* metadata`	コンテキスト・メタデータの配列へのポインター

カウンター・インスタンスを作成し、カウンターデータを送信するには、以下を使用します。

```
__itt_counter_create_v3(__itt_domain* domain, const char* name,
__itt_metadata_type type);
__itt_counter_set_value_v3(__itt_counter counter, void *value_ptr);
```

使用例

この例では、SSD NVMe デバイスのランダム読み取り操作メトリックを測定するコンテキスト・メタデータを含むカウンターを作成します。

```
#include "ittnotify.h"
#include "ittnotify_types.h"

// ドメインとカウンターを作成:
__itt_domain* domain =
 __itt_domain_create("ITT API collected data");
__itt_counter counter_read_op =
 __itt_counter_create_v3(domain, "Read Operations", __itt_metadata_u64);
__itt_counter counter_read_mb =
 __itt_counter_create_v3(domain, "Read Megabytes", __itt_metadata_u64);
```

```
__itt_counter counter_spin_time =
 __itt_counter_create_v3(domain, "Spin Time", __itt_metadata_u64);

// コンテキスト・メタデータを作成
__itt_context_metadata metadata_read_op[] = {
       {__itt_context_name, "Reads"},
       {__itt_context_device, "NVMe SSD Intel DC 660p"},
       {__itt_context_units, "Operations"},
       {__itt_context_pci_addr, "0001:10:00.1"},
       {__itt_context_latency_flag, &true_flag}
};
__itt_context_metadata metadata_read_mb[] = {
       {__itt_context_name, "Read"},
       {__itt_context_device, "NVMe SSD Intel DC 660p"},
       {__itt_context_units, "MB"},
       {__itt_context_pci_addr, "0001:10:00.1"},
       {__itt_context_bandwidth_flag, &true_flag}
};
__itt_context_metadata metadata_spin_time[] = {
       {__itt_context_name, "Spin Time"},
       {__itt_context_device, "NVMe SSD Intel DC 660p"},
       {__itt_context_units, "ms"},
       {__itt_context_tid, &thread_id}
};

// コンテキスト・メタデータをカウンターにバインド:
__itt_bind_context_metadata_to_counter(counter_read_op, n, metadata_read_op);
__itt_bind_context_metadata_to_counter(counter_read_mb, n, metadata_read_mb);
__itt_bind_context_metadata_to_counter(counter_spin_time, n,
metadata_spin_time);

while(1)
{
    // 収集したデータを取得:
    uint64_t read_op   = get_user_read_operation_num();
    uint64_t read_mb   = get_user_read_megabytes_num();
    uint64_t spin_time = get_user_spin_time();

    // 収集したデータをダンプ:
    __itt_counter_set_value_v3(counter_read_op, &read_op);
    __itt_counter_set_value_v3(counter_read_mb, &read_mb);
    __itt_counter_set_value_v3(counter_spin_time, &spin_time);
```

```
}
```

カウンター API

カウンターとは、ハードウェアまたはソフトウェアの動作に関するユーザー定義の特性またはメトリックであり、実行状況の内訳情報を収集するために使用されます。カウンターを使用して情報をタスク、イベント、マーカーと関連付けることもできます。

例えば、システムオンチップ（SoC）開発では、SoC のさまざまな部分を表す複数のカウンターを使用して、ハードウェア特性をカウントできます。

カウンター・オブジェクトを定義および作成

使用するプリミティブ:

```
__itt_counter __itt_counter_create(const char *name, const char *domain);

__itt_counter __itt_counter_createA(const char *name, const char *domain);

__itt_counter __itt_counter_createW(const wchar_t *name, const wchar_t
*domain);

__itt_counter __itt_counter_create_typed(const char *name, const char *domain,
__itt_metadata_type type);

__itt_counter __itt_counter_create_typedA(const char *name, const char *domain,
__itt_metadata_type type);

__itt_counter __itt_counter_create_typedW(const wchar_t *name, const wchar_t
*domain, __itt_metadata_type type);

__itt_counter __itt_counter_create_v3(__itt_domain* domain, const char*
name,__itt_metadata_type type);
```

カウンター名とドメイン名を指定する必要があります。特殊なデータのタイプをロードするには、カウンタータイプを指定します。デフォルトのカウンタータイプは `uint64_t` です。

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	domain	カウンタードメイン

タイプ	パラメーター	説明
[in]	name	カウンター名
[in]	type	カウンタータイプ

インクリメント/デクリメントするカウンター値

使用するプリミティブ:

```
void __itt_counter_inc (__itt_counter id);
void __itt_counter_inc_delta(__itt_counter id, unsigned long long value);
void __itt_counter_dec(__itt_counter id);
void __itt_counter_dec_delta(__itt_counter id, unsigned long long value);
```

注

これらのプリミティブは uint64 カウンターにのみ適用されます。

カウンター値を直接設定します

以下を使用:

```
void __itt_counter_set_value(__itt_counter id, void *value_ptr);
void __itt_counter_set_value_v3(__itt_counter counter, void *value_ptr);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	id	カウンター ID
[in]	value_ptr	カウンター値

既存のカウンターを削除します

以下を使用:

```
void __itt_counter_destroy(__itt_counter id);
```

使用例

この例は、温度とメモリー使用メトリックを測定するカウンターを作成します。

```
#include "ittnotify.h"
```

```
__itt_counter temperatureCounter = __itt_counter_create("Temperature",
"Domain");
__itt_counter memoryUsageCounter = __itt_counter_create("Memory Usage",
"Domain");
unsigned __int64 temperature;

while (...)
{
    ...
    temperature = getTemperature();
    __itt_counter_set_value(temperatureCounter, &temperature);


    __itt_counter_inc_delta(memoryUsageCounter, getAllocatedMemSize());
    __itt_counter_dec_delta(memoryUsageCounter, getDeallocatedMemSize());
    ...
}

__itt_counter_destroy(temperatureCounter);
__itt_counter_destroy(memoryUsageCounter);
```

ドメイン API

ドメイン は、プログラム内のモジュールやライブラリーごとにトレースデータのタグ付けを可能にします。ドメインには一意の文字列を指定します。

各ドメインは、`__itt_domain` 構造体で定義され、コード内の ITT API 呼び出しによってタグ付けできます。

出力トレース・キャプチャー・ファイルに収集されるインストルメントのサブセットをフィルター処理するため、アプリケーション内の特定のドメインを選択的に有効または無効にできます。

ドメインを無効にするには、フラグフィールドを 0 に設定します。この操作を行うと、他のコード部分に影響を与えることなく、特定のドメインのトレースが無効になります。ドメインを無効にするオーバーヘッドは、単一の `if` 文でのチェックのみです。

ドメインを作成するには、次のプリミティブを使用します:

```
__itt_domain *ITTAPI __itt_domain_create ( const char *name);
```

ドメイン名の作成には、URI の命名規則に従ってください。例えば、“com.my_company.my_application” は

ドメイン名として許容される形式です。ドメインのセットは、アプリケーションの実行時間を通じて静的であることが期待されます。したがって、ドメインを破棄する仕組みは存在しません。

プロセス内のどのスレッドでも、そのドメインを作成したスレッドに関係なく、コード内のあらゆるドメインにアクセスできます。この呼び出しはスレッドセーフです。

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	name	ドメインの名前

使用例

```
#include "ittnotify.h"


__itt_domain* pD = __itt_domain_create(L"My Domain" );
pD->flags = 0; /* ドメインを無効化*/
```

イベント API

イベント API を使用して以下を行います:

- アプリケーション内で特定のイベントが発生するタイミングを監視します
- コード内の特定の部分を実行するのにかかる時間を計測します。

アプリケーションにアノテーションを追加して、注目するイベントが発生する領域を区分します。この解析の実行後、[タイムライン] ペインでマークしたイベントを観察できます。

このイベント API は、再開時に動作するスレッド単位の関数です。この関数はこのポーズ状態では動作しません。

注

- Windows* プラットフォームでは、文字列を渡すワイド文字バージョンの API を使用する Unicode を定義できます。これらの文字列は、内部では ASCII 文字列に変換されます。
- Linux* プラットフォームでは、API は 1 つのみです。

使用するプリミティブ	説明
`__itt_event __itt_event_create(const __itt_char *name, int namelen );`	指定する名前と長さでイベントタイプを作成します。この API は、イベントタイプへのハンドルを返します。このハンドルは、以下のイベント開始 API とイベント終了 API にパラメーターとして渡す必要があります。namelen パラメーターは、文字数で名前の長さ

使用するプリミティブ	説明
	を指定します。
`int __itt_event_start(__itt_event event);`	イベントタイプのハンドルを使用してこの API を呼び出し、イベントのインスタンスを登録します。このイベントの開始は、[タイムライン] ペインにチェックマークとして表示されます。
`int __itt_event_end(__itt_event event);`	`__itt_event_start()` の呼び出し後にこの API を呼び出して、開始から終了までの期間をチェックマークとしてイベントを表示します。この API が呼び出されない場合、イベントは [タイムライン] ペインに単独のチェックマークとして表示されます。

利用ガイドライン

- `__itt_event_end()` は常に直前の `__itt_event_start()` とペアになります。それ以外では、_itt_event_end() 呼び出しは先行する対にならない `__itt_event_start()` に対応付けられます。干渉するイベントはすべて入れ子にされます。
- 同一タイプまたは異なるタイプのユーザーイベントを相互に入れ子にできます。イベントが入れ子になる場合、その時間は最も深く入れ子になったユーザーイベント領域でのみ費やされたと見なされます。
- 異なる ITT API イベントをオーバーラップできます。イベントが重複する場合、時間は `__itt_event_start()` を持つイベント領域でのみ費やされたと見なされます。一致しない `__itt_event_end()` 呼び出しは無視されます。

注

結果にイベントとユーザータスクを表示するには、(注目する事前定義された解析を基に) [カスタム解析] を作成して、解析設定の**[ユーザータスク、イベントおよびカウンターを解析]** チェックボックスをオンにします。

使用例: 単独のイベントを作成およびマーク

`__itt_event_create` API は、`__itt_event_start` API を使用してユーザーイベントをマークするために使用する新しいイベントハンドラーを返します。この例では、2 つのイベント・タイプ・ハンドラーが作成され、異なるタイプのイベントを追跡する開始点を設定するために使用されます。

```
#include "ittnotify.h"

__itt_event mark_event = __itt_event_create( "User Mark", 9 );
__itt_event frame_event = __itt_event_create( "Frame Completed", 15 );
...
__itt_event_start( mark_event );
...
for( int f ; f<number_of_frames ; f++ ) {
```

```
  ...
  __itt_event_start( frame_event );
}
```

使用例: イベント範囲を作成およびマーク

次の例のように、`__itt_event_start` API の後に `__itt_event_end` API を続けてイベント領域を定義します:

```
#include "ittnotify.h"

__itt_event render_event = __itt_event_create( "Rendering Phase", 15 );
...
for( int f ; f<number_of_frames ; f++ ) {
  ...
  do_stuff_for_frame();
  ...
  __itt_event_start( render_event );
  ...
  do_rendering_for_frame();
  ...
  __itt_event_end( render_event );
  ...
}
```

形式指定メタデータ API

形式指定メタデータ API を使用すると、効率的な文字列フォーマット機能を利用して、形式指定の文字列データをタスクに添付できます。この API は、`printf` スタイルの書式設定機能を提供することで、基本的なメタデータ API を拡張します。

形式指定メタデータ API は、スレッドごとに機能する関数であり、プロファイルが再開された状態でのみ動作します。

API 関数

現在のタスクに形式指定メタデータを追加するには、以下のプリミティブを使用します:

```
void __itt_formatted_metadata_add(const __itt_domain *domain,
                                  __itt_string_handle *format_handle,
                                  ...);
```

重複するタスクに形式指定メタデータを追加するには、以下のプリミティブを使用します:

```
void __itt_formatted_metadata_add_overlapped(const __itt_domain *domain,
                                             __itt_id taskid,
                                             __itt_string_handle *format_handle,
                                             ...);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`__itt_domain* domain`	メタデータドメイン
[in]	`__itt_id taskid`	タスク ID（重複処理を行う場合にのみ必須）
[in]	`__itt_string_handle* format_handle`	`printf` スタイルの書式指定子を含む書式文字列を格納する文字列ハンドル
[in]	`...`	書式文字列内の書式指定子に対応する可変個数の引数

メタデータの可視化と解析

書式化されたメタデータは、インテル® VTune™ プロファイラーにおける視覚化と解析でいくつかの利点をもたらします:

- **タイムライン・ツールチップ:** タイムライン・ビューでタスクにカーソルを合わせると、ツールチップにメタデータが表示され、コンテキストに応じた実行時情報がもたらされます。
- **[ボトムアップ] ビューのグループ化:** 書式化されたメタデータは、[ボトムアップ] ビューにおけるグループ化基準として使用でき、メタデータの値に基づいてタスクを整理および解析できます。
- **グループ化をカスタマイズ:** ユーザーは、タスクをメタデータの値に基づいてグループ化することで、さまざまな実行コンテキストにおけるパターンやパフォーマンス特性を特定できます。
- **書式指定子における角括弧の表記法:** 書式文字列の中に角括弧で囲まれたフォーマット指定子（例: `[%s]`）を含めると、インテル® VTune™ プロファイラーはこれらを特別なグループ識別子として扱います。

利用ガイドライン

- **サポートされる書式指定子:** `%s`、`%ls`、`%d`、`%u`、`%hd`、`%hu`、`%ld`、`%lu`、`%lld`、`%llu`、`%f`、`%lf`
- **通常タスク:** 現在実行中のタスクに関連付けられたメタデータには、`__itt_formatted_metadata_add` を使用します
- **重複タスク:** 重複するタスク・インスタンスには、特定のタスク ID を指定して `__itt_formatted_metadata_add_overlapped` を使用します
- **タスクごとにメタデータ呼び出しは 1 回に限定する** - 同じタスクに対して `__itt_formatted_metadata_add` を複数回呼び出すと、処理が正しく行われない可能性があります。
- 最適なパフォーマンスを得るには、メタデータの追加頻度とサイズを制限します

- 角括弧で囲まれたフォーマット指定子（例: `[%s]`）を使用すると、インテル® Vtune™ プロファイラーの解析ビューで追加のグループ化オプションを作成できます
- 関数引数は API 呼び出し中に処理されます
- 文字列引数の最大長は 256 文字です

使用例

```
#include "ittnotify.h"

__itt_domain* domain = __itt_domain_create("FileProcessor");
__itt_string_handle* operation_format = __itt_string_handle_create("Operation:
[%s] on file %s");
__itt_string_handle* performance_format =
__itt_string_handle_create("Performance: %d bytes in %.2f ms");

void process_file(const char* filename) {
    __itt_task_begin(domain, __itt_null, __itt_null,
__itt_string_handle_create("process_file"));

    __itt_formatted_metadata_add(domain, operation_format, "file_processing",
filename);

    __itt_task_begin(domain, __itt_null, __itt_null,
__itt_string_handle_create("read_file"));

    int bytes_read = 1024;
    double read_time = 15.5;
    __itt_formatted_metadata_add(domain, performance_format, bytes_read,
read_time);

    __itt_task_end(domain);

    __itt_task_begin(domain, __itt_null, __itt_null,
__itt_string_handle_create("transform_data"));

    __itt_formatted_metadata_add(domain, operation_format, "data_transform",
filename);

    __itt_task_end(domain);
    __itt_task_end(domain);
}
```

```
int main() {
    process_file("document.txt");
    process_file("image.jpg");
    return 0;
}
```

フレーム API

フレーム API を使用して、コード内の目的の場所に呼び出しを挿入し、フレームごとのパフォーマンスを解析します。フレームは、フレーム開始点と終了点の間の時間範囲をとして定義されます。インテル® VTune™ プロファイラーでフレームを表示すると、通常のタスクデータとこのデータを視覚的に分離した別トラックで表示されます。

このフレーム API は、再開時に動作するプロセス単位の関数です。この関数はこのポーズ状態では動作しません。

フレーム解析を実行します:

- DirectX* レンダリングを使用する Windows* のゲーム・アプリケーション解析。
- 繰り返し計算を行うグラフィックス・アプリケーションの解析。
- トランザクションごとに処理を解析してパフォーマンスが低下する入力ケースを検出。

フレームは、経過時間のオーバーラップしない領域を表します。フレームは本来グローバルであるため、特定のスレッドに関連付けられません。ITT API は、コードフレームの解析と解析データを提供します。

コードにフレームを API をインクルード

`__itt_domain_create()` 関数を使用してドメイン・インスタンスを作成します:

```
__itt_domain *ITTAPI __itt_domain_create ( const char *name );
```

ドメイン名を作成する際は、URI の命名規則に従ってください。例えば、“com.my_company.my_application” は許容される形式です。ドメインのセットは、アプリケーションの実行時間を通じて静的であることが期待されます。したがって、ドメインを破棄する仕組みは存在しません。

プロセス内のどのスレッドでも、ドメインを作成したスレッドに関係なく、コード内のあらゆるドメインにアクセスできます。この呼び出しはスレッドセーフです。

フレーム・インスタンスの始まりを定義します。`__itt_frame_begin_v3` は、`__itt_frame_end_v3` とペアで使用する必要があります:

```
void __itt_frame_begin_v3(const __itt_domain *domain, __itt_id *id);
```

同じ ID を使用して `__itt_frame_begin_v3` を呼び出しても、同一 ID で `__itt_frame_end_v3` が呼び出すまで無視されます。

タイプ	パラメーター	説明
[in]	domain	このフレーム・インスタンスのドメイン
[in]	id	このフレーム・インスタンスのインスタンス ID。NULL を指定できます。*id* パラメーターとして、NULL を指定した次の `__itt_frame_end_v3` 呼び出しで、フレームの終わりを指定します。

フレーム・インスタンスの終わりを定義します。`__itt_frame_begin_v3` は、`__itt_frame_end_v3` とペアで使用する必要があります。ID を指定して最初に `__itt_frame_end_v3` を呼び出すと、フレームが終了します。同じ ID による連続した呼び出しは、一致する `__itt_frame_begin_v3` 呼び出しがない場合と同様に無視されます:

```
void __itt_frame_end_v3(const __itt_domain *domain, __itt_id *id);
```

タイプ	パラメーター	説明
[in]	domain	このフレーム・インスタンスのドメイン
[in]	id	このフレーム・インスタンスのインスタンス Id、または現在のインスタンスでは NULL です

注

ハードウェア・イベントベース・サンプリング収集ベースの解析タイプでは、64 個の異なるフレームドメインに制限されています。

利用ガイドライン

- フレーム API を使用して、フレームの開始点と終了点を指示します。フレームの開始点と終了点の間の範囲をフレームとして示します。
- インテル® VTune™ プロファイラーは、`__itt_frame_end_v3()` と `__itt_frame_begin_v3()` 間の時間/サンプルはプログラム単位に関連付けないため、ビューポイントには `[Unknown]` として表示されます。
- 同じドメインに連続した `__itt_frame_begin_v3` 呼び出しがある場合、`__itt_frame_end_v3` / `__itt_frame_begin_v3` のペアとして扱われます。
- 同じドメインの再帰/入れ子/オーバーラップ・フレームは許されません。
- `__itt_frame_begin_v3()` と `__itt_frame_end_v3()` 呼び出しは、異なるスレッドからも実行できます。
- フレーム API を呼び出す最大レートの推奨値は毎秒 1000 フレームです。フレームレートを高くすると、メモリー消費量が増えてファイナライズに時間がかかります。

使用例

次の例は、フレーム API を使用して、指定したコード領域の経過時間を取得します。

```
#include "ittnotify.h"

__itt_domain* pD = __itt_domain_create( L"My Domain" );

pD->flags = 1; /* ドメインを有効にします */

for (int i = 0; i < getItemCount(); ++i)
{
  __itt_frame_begin_v3(pD, NULL);
  do_foo();
  __itt_frame_end_v3(pD, NULL);
}

//...

__itt_frame_begin_v3(pD, NULL);
do_foo_1();
__itt_frame_end_v3(pD, NULL);

//...

__itt_frame_begin_v3(pD, NULL);
do_foo_2();
__itt_frame_end_v3(pD, NULL);
```

ヒストグラム API

ヒストグラム API を使用して、インテル® VTune™ プロファイラーで任意のデータをヒストグラム（分布図）形式で表示するヒストグラムを定義します。

ヒストグラムは、相互比較のため個々のユニットで分割可能な統計を表示する場合に有効です。

ヒストグラム API を使用すると、以下のことが可能です:

- 負荷分散の追跡
- リソース利用率の追跡
- ワーカーノードのオーバーサブスクリプションと未使用を特定

プロセス内のどのスレッドでも、そのドメインを作成したスレッドに関係なく、コード内のあらゆるドメインにアクセスできます。ヒストグラム API 呼び出しはスレッドセーフです。

ヒストグラムを定義して作成

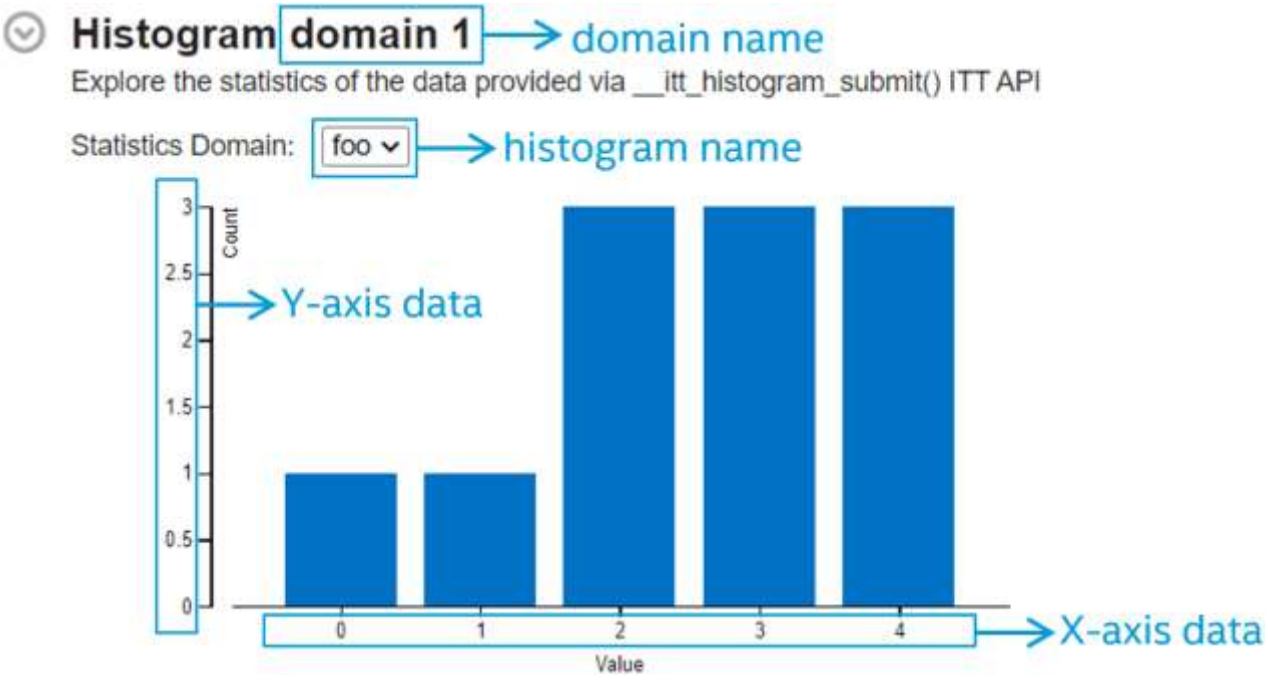
ヒストグラムを作成する前に、ITT API ドメインを作成する必要があります。このドメインへのポインターがプリミティブに渡されます。

ドメイン名は、インテル® VTune™ プロファイラーの結果の **[サマリー]** タブにある分布図の見出しに表示されます。

1 つのドメインで、複数の分布図を組み合わせることができます。ただし、分布図の名称は同一ドメインでは同じである必要があります。

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`domain`	呼び出しを制御するドメイン
[in]	`name`	ヒストグラム名
[in]	`x_axis_type`	X 軸データのタイプ
[in]	`y_axis_type`	Y 軸データのタイプ



プリミティブ:

使用するプリミティブ	説明

`__itt_histogram* _itt_histogram_create(`
`  __itt_domain* domain,`
`  const char* name,`
`  __itt_metadata_type x_axis_type,`
`  __itt_metadata_type y_axis_type);`
	Linux* および Android* で、指定されたドメイン、名前、データタイプの分布図を作成します。

`__itt_histogram*`
`_itt_histogram_createA(`
`  __itt_domain* domain,`
`  const char* name,`
`  __itt_metadata_type x_axis_type,`
`  __itt_metadata_type y_axis_type);`
	Windows* で、ASCII 文字列（`char`）に対し、指定されたドメイン、名前、データタイプの分布図を作成します。

`__itt_histogram*`
`_itt_histogram_createW(`
`  __itt_domain* domain,`
`  const wchar_t* name,`
`  __itt_metadata_type x_axis_type,`
`  __itt_metadata_type y_axis_type);`
	Windows* で UNICODE 文字列（`wchar_t`）に対し、指定されたドメイン、名前、データタイプの分布図を作成します。

分布図にデータを送信

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`histogram`	データを送信する分布図のインスタンス
[in]	`length`	送信された軸データ配列の要素数
[in]	`x_axis_data`	X 軸のデータを含む配列（`NULL` でも可）。`x_axis_data` が `NULL` の場合、インテル® VTune™ プロファイラーは `y_axis_data` 配列のインデックスを使用します。
[in]	`y_axis_data`	Y 軸のデータを含む配列

プリミティブ:

使用するプリミティブ	説明
`void _itt_histogram_submit(` `  __itt_histogram* histogram,`	選択されたヒストグラムのインスタンスのユーザー統計情報を送信します。2 次元平面上の点の座標と同じように、

使用するプリミティブ	説明
size_t length, void* x_axis_data, void* y_axis_data);	Y 軸の配列データは X 軸の配列データに対応付けられています。ワークロードの実行中に送信されたデータは、このプリミティブのすべての呼び出しで 1 つの共通の分布図に集約されます。収集にかかるコストを削減するには、データ送信の間隔を効率的に設定することが重要です。

使用例

次の例では、ワーカースレッドの統計情報を保存する分布図を作成しています:

```
#include "ittnotify.h"
#include "ittnotify_types.h"



void submit_stats()
{
    // ドメインを作成
    __itt_domain* domain = __itt_domain_create("Histogram statistics domain");


    // 分布図を作成
    __itt_histogram* histogram = __itt_histogram_create(domain, "Worker TID
13454", __itt_metadata_u64, __itt_metadata_u64);


    // プロファイル・データを使って統計配列を埋める:
    uint64_t* x_stats, y_stats;
    size_t array_size;
    get_worker_stats(x_stats, y_stats, array_size);


    // ヒストグラム統計を送信:
    __itt_histogram_submit(histogram, array_size, x_stats, y_stats);
}
```

ロードモジュール API

コード内でロードモジュール API を使用して、インテル® VTune™ プロファイラーがトラックできない場所にロードされたモジュールを解析できます。たとえば、これにより、通常はコードの可視性がない分離された環境で実

行されるコードを解析できるようになります。この API を使用して、インテル® VTune™ プロファイラーによる解析のためアドレス空間内のモジュール位置を明示的に設定できます。

使用するプリミティブ	説明
`void __itt_module_loadW(void* start_addr, void* end_addr, const wchar_t* path);`	モジュールの再配置後にこの関数を呼び出します。モジュールの新しい開始アドレスと終了アドレス、およびローカルドライブ上のモジュールへの完全パスを指定します。
`void __itt_module_loadA(void* start_addr, void* end_addr, const char* path);`	モジュールの再配置後にこの関数を呼び出します。モジュールの新しい開始アドレスと終了アドレス、およびローカルドライブ上のモジュールへの完全パスを指定します。
`void __itt_module_load(void* start_addr, void* end_addr, const char* path);`	モジュールの再配置後にこの関数を呼び出します。モジュールの新しい開始アドレスと終了アドレス、およびローカルドライブ上のモジュールへの完全パスを指定します。

使用例

```
#include "ittnotify.h"

__itt_module_load(relocatedBaseModuleAddress, relocatedEndModuleAddress,
"/path/to/dynamic/library.so");
```

マーカー API

マーカーとは、特定のプロセスやスレッドに関連付けられる、あるいはグローバルスコープで指定される、タイムライン上の瞬間的なイベントです。

マーカーを作成するには、次のプリミティブを使用します:

```
void __itt_marker(const __itt_domain *domain, __itt_id id,
                  __itt_string_handle *name, __itt_scope scope);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	domain	マーカードメイン
[in]	name	マーカー名

タイプ	パラメーター	説明
[in]	id	オプションのパラメーター。マーカー ID、または `__itt_null`。ドメインが異なるマーカーは、同じ ID を持つことはできません。
[in]	scope	マーカーの適用範囲：プロセス、スレッド、およびグローバル

メモリー割り当て API

インテル VTune™ プロファイラーは、`malloc` と類似したヒープ管理機能のセマンティクスを識別するのに有用な API を提供します。

これらの API を使用してコードにアノテーションを追加すると、インテル® VTune™ プロファイラーはメモリーアクセス解析の一部としてメモリー・オブジェクトを正確に判別できます。

利用ガイドライン

メモリー割り当て API を使用する際は、以下のガイドラインに従ってください:

- ルーチンのラッパー関数を作成します。それらの関数の中に、`__itt_heap_*_begin` および `__itt_heap_*_end` 呼び出しを挿入します。
- アプリケーションが `__itt_heap_function_create` を呼び出す場合、`allocate/free` 関数のペアごとに固有のドメインを割り当てます。これにより、インテル® VTune™ プロファイラーは、すべての `allocate` 関数呼び出しに対応する `free` 関数の呼び出しを確認できます。
- すべての `allocate` 関数と `free` 関数の前後にアノテーションを追加します。
- 同じスタックフレームからすべての関数ペアを呼び出します。そうしないと、インテル® VTune™ プロファイラーは、例外が発生してメモリー割り当てが失敗したとみなします。
- 対応する `begin` 関数なしで `end` 関数を呼び出してはなりません。

コードでメモリー割り当て API を使用する

操作	説明

`typedef void* __itt_heap_function;`
 	
`__itt_heap_function __itt_heap_function_create( const __itt_char* <name>, const __itt_char* <domain> );`
	`begin` および `end` 呼び出しとドメインに一致するハンドルタイプを宣言します。
	`name` = アノテーションしたい関数の名前。
	`domain` = 一致する関数セットを識別する文字列。
	例えば、`alloc_my_structs`、`free_my_structs`、および `realloc_my_structs` など、`my_struct` で動作する 3 つの関数がある場合、同じドメインを 3 つの `__itt_heap_function_create()` 呼び出しに渡します。

操作	説明

`void __itt_heap_allocate_begin( __itt_heap_function <h>, size_t <size>, int <initialized> );`
 	
`void __itt_heap_allocate_end( __itt_heap_function <h>, void**, size_t <size>, int <initialized> );`
	割り当て関数を特定します。
	h = この関数名が `__itt_heap_function_create()` に渡されたときに返されるハンドル。
	`size` = 要求されたメモリー領域のバイト単位のサイズ。
	`initialized` = このルーチンによってメモリー領域が初期化されるかどうかを示すフラグ。
	`addr` = この関数が割り当てられたメモリー領域のアドレスへのポインター。割り当てが失敗した場合は 0 になります。

`void __itt_heap_free_begin( __itt_heap_function <h>, void* );`
 	
`void __itt_heap_free_end( __itt_heap_function <h>, void* );`
	割り当て解除関数を特定します。
	`h` = この関数名が `__itt_heap_function_create()` に渡されたときに返されるハンドル。
	`addr` = この関数が解放するメモリー領域のアドレスへのポインター。

`void __itt_heap_reallocate_begin( __itt_heap_function <h>, void*, size_t <new_size>, int <initialized> );`
 	
`void __itt_heap_reallocate_end( __itt_heap_function <h>, void*, void** <new_addr>,`
	再割り当て関数を特定します。
	注意: `itt_heap_reallocate_end()` は、メモリーが返されなくとも要求後に呼び出す必要があります。インテル® VTune™ プロファイラーは、C ランタイムの `realloc` セマンティクスを想定しています。
	`h` = この関数名が `__itt_heap_function_create()` に渡されたときに返されるハンドル。
	`addr` =この関数が再割り当するメモリー領域のアドレスへのポインター。`addr` が NULL の場合、インテル® VTune™ プロファイラーはこれを割り当てのように扱います。
	`new_addr` = 再割り当てされたメモリー領域のアドレスを保持するポインターへのポインター。

操作	説明
size_t <new_size>, int <initialized>);	size = 要求されたメモリー領域のバイト単位のサイズ。 new_size が 0 の場合、インテル® VTune™ プロファイラーはこれを解放されたように扱います。

使用例: ヒープ割り当て

```
#include <ittnotify.h>

void* user_defined_malloc(size_t size);
void user_defined_free(void *p);
void* user_defined_realloc(void *p, size_t s);



__itt_heap_function my_allocator;
__itt_heap_function my_reallocator;
__itt_heap_function my_freer;

void* my_malloc(size_t s)
{
    void* p;

    __itt_heap_allocate_begin(my_allocator, s, 0);
    p = user_defined_malloc(s);
    __itt_heap_allocate_end(my_allocator, &p, s, 0);

    return p;
}

void my_free(void *p)
{
    __itt_heap_free_begin (my_freer, p);
    user_defined_free(p);
    __itt_heap_free_end (my_freer, p);
}

void* my_realloc(void *p, size_t s)
{
    void *np;

    __itt_heap_reallocate_begin (my_reallocator, p, s, 0);
```

```
    np = user_defined_realloc(p, s);
    __itt_heap_reallocate_end(my_reallocator, p, &np, s, 0);

    return(np);
}

// ユーザー定義のアロケーターを呼び出す前に、
// 必ずこの初期化ルーチンを呼び出します。
void init_itt_calls()
{
    my_allocator = __itt_heap_function_create("my_malloc", "mydomain");
    my_reallocator = __itt_heap_function_create("my_realloc", "mydomain");
    my_freer = __itt_heap_function_create("my_free", "mydomain");
}
```

メタデータ API

メタデータとは、タスク、スレッド、プロセスなどに付加できる追加情報、または汎用データのことであり、メタデータには、タイプ、名前、値があります。値のエンコードは、メタデータのタイプによって異なります。このエンコードには、文字列データ、あるいは複数の整数値または浮動小数点数値が含まれる場合があります。

メタデータを作成するには、次のプリミティブを使用します:

```
void __itt_metadata_add(const __itt_domain *domain, __itt_id id,
                    __itt_string_handle *key,
                    __itt_metadata_type type, size_t count, void *data);

void __itt_metadata_str_addA(const __itt_domain *domain, __itt_id id,
                         __itt_string_handle *key,
                         const char *data, size_t length);

void __itt_metadata_str_addW(const __itt_domain *domain, __itt_id id,
                         __itt_string_handle *key,
                         const wchar_t *data, size_t length);

void __itt_metadata_add_with_scope(const __itt_domain *domain, __itt_scope scope,
                               __itt_string_handle *key, __itt_metadata_type type,
                               size_t count, void *data);

void __itt_metadata_str_add_with_scopeA(const __itt_domain *domain,
                                __itt_scope scope,
                                __itt_string_handle *key,
```

```
                                const char *data, size_t length);

void __itt_metadata_str_add_with_scopeW(const __itt_domain *domain,
                                __itt_scope scope,
                                __itt_string_handle *key,
                                const wchar_t *data, size_t length);
```

次の表は、メタデータ API プリミティブで使用するパラメーターを定義します。

タイプ	パラメーター	説明
[in]	__itt_domain* domain	メタデータドメイン
[in]	__itt_scope scope	メタデータの適用範囲：タスク、スレッド、プロセス、およびグローバル。スコープが定義されていない場合、メタデータはスレッド内の最後のタスクに属します。
[in]	__itt_string_handle* name	メタデータ名
[in]	__itt_metadata_type type	メタデータ・タイプ。数値メタデータにのみ使用されます。
[in]	size_t count	数値メタデータ項目数 `[in] size_t length`
[in]	size_t length	メタデータ文字列のシンボル数
[in]	void *data const char *data const wchar_t *data	実際のメタデータ（数値または文字列の配列）

リレーション API

リレーション API は、タスクなどの 2 つの名前付きインスタンスを、適切な関連属性で関連付けます。実際にはインスタンスが作成される前後で、関連付けを追加することができます。これらの関係は個々のインスタンスとは独立しています。

複数のタスクを論理的にグループ化するには、さまざまなタイプの関係を使用できます:

```
void ITTAPI __itt_relation_add(const __itt_domain *domain, __itt_id head,
                           __itt_relation relation, __itt_id tail);

void ITTAPI __itt_relation_add_ex(const __itt_domain *domain, __itt_clock_domain*
clock_domain,
                               unsigned long long timestamp, __itt_id head,
                               __itt_relation relation, __itt_id tail);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`__itt_domain* domain`	リレーション・ドメイン
[in]	`__itt_relation relation`	2 つの名前付きインスタンス間のユーザー定義の論理的関係
[in]	`__itt_id head`	メタデータ名
[in]	`__itt_id tail`	2 つの名前付き関連インスタンスの ID `size_t count`
[in]	`__itt_clock_domain* clock_domain`	ユーザー定義のクロックドメイン
[in]	`unsigned long long timestamp`	対応するクロックドメインのユーザー定義タイムスタンプ

文字列ハンドル API

多くの API 呼び出しでは、API オブジェクトを識別する名前が必要です。文字列ハンドルはこれらの名前へのポインターです。文字列ハンドルを使用すると、実行時に名前付きオブジェクトの効率良い処理が可能になります。これらのハンドルによって、収集されたトレースデータがコンパクトになります。

文字列に関連付けるハンドル値を作成して返すには、次のプリミティブを使用します:

```
__itt_string_handle* __itt_string_handle_create(const char *name);
```

同じ名前の `__itt_string_handle_create` の連続した呼び出しは、同じ値を返します。文字列ハンドルのセットは、アプリケーションの実行中は静的な状態を維持することが期待されます。したがって、文字列ハンドルを破棄する仕組みは存在しません。プロセス内のどのスレッドでも、文字列ハンドルを作成したスレッドに関係なく、コード内のあらゆる文字列ハンドルにアクセスできます。この呼び出しはスレッドセーフです。

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	`name`	入力文字列

タスク API

タスクは、特定のスレッドで実行されるワークの論理単位です。タスクは入れ子にできます。そのため、通常、タスクは関数、スコープ、または `switch` 文の `case` ブロックに対応します。

タスクをスレッドに割り当てるためタスク API を使用します

タスク API では、スレッドは以下の操作を実行できません:

- タスク切り替えとは、スレッドが現在のタスクを中断し、別のタスクに切り替えることです。
- タスクスチールとは、あるスレッドがタスクを別のスレッドに移行することです。

タスクのインスタンスは、ある期間に特定のスレッドが実行するワークに相当します。タスクは同一スレッド内で `__itt_task_begin()` と `__itt_task_end()` で囲むことで定義します。
形式指定メタデータ API を使用することで、タスクにフォーマットされたメタデータを追加し、タスクの表現力を向上できます。これにより、インテル® VTune™ プロファイラーの解析ビューに表示される情報に、printf スタイルの書式設定でコンテキスト情報を追加することができます。

タスクには単純なものもあれば、複数のタスクが重複する場合もあります。

単純なタスクは、組み込み実行という概念を暗黙のうちにサポートします。`__itt_task_end()` 関数は、直前の `__itt_task_begin()` の呼び出しを完了させます。例えば、以下のメタコードは有効なシーケンスであり、“a” タスクの実行時間には “b” タスクの実行時間が含まれます:

```
__itt_task_begin(a);
__itt_task_begin(b);
__itt_task_end(b);
__itt_task_end(a);
```

重複するタスクの実行領域は、互いに干渉する可能性があります。例えば、以下のメタコードは有効なシーケンスです。“a” タスクの後に開始された “b” タスクは、“a” タスクの完了と同時に終了できます:

```
__itt_task_begin_overlapped(a);
__itt_task_begin_overlapped(b);
__itt_task_end_overlapped(a);
__itt_task_end_overlapped(b);
```

タスク API 関数

スレッド上でシンプルなタスク・インスタンスを作成するには、以下の関数を使用します:

```
void ITTAPI __itt_task_begin(const __itt_domain *domain, __itt_id taskid,
```

```
                          __itt_id parentid, __itt_string_handle *name);

void ITTAPI __itt_task_begin_fn (const __itt_domain *domain,__itt_id taskid,
                          __itt_id parentid, void* address);

void ITTAPI __itt_task_end (const __itt_domain *domain);
```

異なるクロックドメインでシンプルなタスク・インスタンスを作成するには、以下の関数を使用します:

```
void ITTAPI __itt_task_begin_ex(const __itt_domain* domain,
                            __itt_clock_domain* clock_domain,
                            unsigned long long timestamp, __itt_id taskid,
                            __itt_id parentid,
                            __itt_string_handle* name);

void ITTAPI __itt_task_begin_fn_ex(const __itt_domain* domain,
                              __itt_clock_domain* clock_domain,
                              unsigned long long timestamp, __itt_id taskid,
                              __itt_id parentid, void* fn);

void ITTAPI _itt_task_end_ex(const __itt_domain* domain,
                              __itt_clock_domain* clock_domain,
                              unsigned long long timestamp);
```

スレッド上で重複実行可能なタスク・インスタンスを作成するには、以下の関数を使用します:

```
void ITTAPI __itt_task_begin_overlapped(const __itt_domain* domain,
                                    __itt_id taskid,
                                    __itt_id parentid, __itt_string_handle* name);

void ITTAPI __itt_task_end_overlapped(const __itt_domain *domain,
                                    __itt_id taskid);
```

これらの関数では、引数 `taskid` は必須です。

異なるクロックドメインで重複するタスク・インスタンスを作成するには、以下の関数を使用します:

```
void ITTAPI __itt_task_begin_overlapped_ex(const __itt_domain* domain,
                                      __itt_clock_domain* clock_domain,
                                      unsigned long long timestamp, __itt_id taskid,
                                      __itt_id parentid, __itt_string_handle* name);
```

```
void ITTAPI __itt_task_end_overlapped_ex(const __itt_domain* domain,
                                         __itt_clock_domain* clock_domain,
                                         unsigned long long timestamp, __itt_id taskid);
```

これらの関数では、引数 `taskid` は必須です。

ITTAPI_itt_task_* 関数のパラメーター

次の表は、タスク API プリミティブで使用するパラメーターを定義します。

タイプ	パラメーター	説明
[in]	`__itt_domain`	タスクのドメイン。
[in]	`__itt_id taskid`	ユーザー定義 ID は、重複するタスク・インスタンスを除き、すべてのタスク・インスタンスでオプションです。`__itt_null` は、未定義のタスク・インスタンスのデフォルト値として使用できます。タスク ID は、タスク・インスタンス間の関連性を定義するのに使用されます。
[in]	`__itt_id parentid`	オプションのパラメーター。タスクが属する親インスタンスの ID、または `__itt_null` です。
[in]	`__itt_string_handle`	タスク文字列のハンドル。
[in]	`void* fn`	名前の代わりに使用できる関数アドレス。例えば、デバッグシンボル情報を使用することで、関数アドレスで関数名を解決することができます。
[in]	`__itt_clock_domain`	ユーザー定義のクロックドメイン。
[in]	`unsigned long long timestamp`	対応するクロックドメインのユーザー定義タイムスタンプ。

使用例

次のコードは、グローバルスコープでドメインと 2 つのタスクを生成します。

```
#include "ittnotify.h"

void do_foo(double seconds);

__itt_domain* domain = __itt_domain_create("MyTraces.MyDomain");
```

```
__itt_string_handle* shMyTask = __itt_string_handle_create("My Task");
__itt_string_handle* shMySubtask = __itt_string_handle_create("My SubTask");


void BeginFrame() {
    __itt_task_begin(domain, __itt_null, __itt_null, shMyTask);
    do_foo(1);
}

void DoWork() {
    __itt_task_begin(domain, __itt_null, __itt_null, shMySubtask);
    do_foo(1);
    __itt_task_end(domain);
}
void EndFrame() {
    do_foo(1);
    __itt_task_end(domain);
}

int main() {
   BeginFrame();
   DoWork();
   EndFrame();
   return 0;
}

#ifdef _WIN32
#include <ctime>

void do_foo(double seconds) {
   clock_t goal = (clock_t)((double)clock() + seconds * CLOCKS_PER_SEC);
   while (goal > clock());
}
#else
#include <time.h>

#define NSEC 1000000000
#define TYPE long

void do_foo(double sec) {
     struct timespec start_time;
     struct timespec current_time;
```

```
    clock_gettime(CLOCK_REALTIME, &start_time);
    while(1) {
        clock_gettime(CLOCK_REALTIME, &current_time);
        TYPE cur_nsec=(long)((current_time.tv_sec-start_time.tv_sec-sec)*NSEC
+
                    current_time.tv_nsec - start_time.tv_nsec);
        if(cur_nsec>=0)
            break;
    }
}
#endif
```

スレッド命名 API

デフォルトでは、アプリケーション内の各スレッドはタイムライン・トラックに表示されます。スレッドにはデフォルトのラベルが使用され、そのラベルは OS のスレッド名を使用するか、プロセス ID とスレッド ID から生成されます。スレッドに分かりやすい名前を付けるには、スレッド命名 API を使用してください。

このスレッド命名 API は、すべての状態（一時停止または再開）で機能するスレッドごとの関数です。この API はスレッドで呼び出す必要があります。

`char` または Unicode 文字列でスレッド名を設定するには、プリミティブを使用します:

```
void __itt_thread_set_name (const __itt_char *name);
```

プリミティブのパラメーター:

タイプ	パラメーター	説明
[in]	name	スレッド名

解析からこのスレッドを除外することを指示します:

```
void __itt_thread_ignore (void);
```

`__itt_thread_ignore()` を呼び出しても、アプリケーションの並行処理には影響しません。この呼び出しの後、現在のスレッドは [タイムライン] ペインに表示されなくなります。

スレッド名が複数設定されると、最後の名前のみが使用されます。

使用例

この例は、特定のスレッドに分かりやすい名前を付け、サービススレッドを無視する方法を示しています。

```
DWORD WINAPI service_thread(LPVOID lpArg)
{
  __itt_thread_ignore();
  // ここにサービスワークを記述。このスレッドは表示されません。
  return 0;
}

DWORD WINAPI thread_function(LPVOID lpArg)
{
  __itt_thread_set_name("My worker thread");
  // ここに注スレッドのワークを記述
  return 0;
}

int main(int argc, char* argv[])
{
  CreateThread(NULL, 0, service_thread, NULL, 0, NULL);
  CreateThread(NULL, 0, thread_function, NULL, 0, NULL);

  return 0;
}
```

ユーザー定義の同期 API

インテル® VTune™ プロファイラーは、複数の Windows* および POSIX* API をサポートしていますが、独自の同期構造を定義することが役立つ場合もあります。インテル® VTune™ プロファイラーは、通常ユーザーが作成したカスタム構造を追跡しません。ただし、同期 API を使用することで、定義した同期構造に関する統計情報を収集できます。

ユーザー定義の同期 API は、スレッドごとに機能する関数であり、プロファイルが再開された状態でのみ動作します。

同期構造は、一般に一連のシグナルとしてモデル化できます。1 つまたは複数のスレッドが、特定のアクションを実行できることを示す別のスレッドグループからのシグナルを待機することがあります。同期 API は、スレッドがシグナルを待機し始めてから、シグナルが到達するまでの追跡を記録します。この情報は、コードをより深く理解するのに役立ちます。この API は、一連のプリミティブとメモリーハンドルを使用して、ユーザー定義の同期オブジェクトに関連する統計を収集します。

注

ユーザー定義同期 API は、スレッド化解析タイプと連携します。

コードでユーザー定義同期 API を使用

次の表は、Windows* または Linux* オペレーティング・システムで利用可能なユーザー定義の同期 API プリミティブについて説明します。

使用するプリミティブ	説明
`void __itt_sync_create( void *addr, const __itt_char *objtype, const __itt_char *objname, int attribute)`	char または Unicode 文字列を使用する同期オブジェクトの作成を登録します。
`void __itt_sync_rename( void *addr, const __itt_char *name)`	作成後、char または Unicode 文字列を使用して同期オブジェクトに名前を割り当てます。
`void __itt_sync_destroy( void *addr)`	破棄されたオブジェクトのライフタイムを追跡します。
`void __itt_sync_prepare( void *addr)`	ユーザー定義の同期オブジェクトでスピンループに入ります。
`void __itt_sync_cancel( void *addr)`	スピン・オブジェクトを取得せずにスピンループを出ます。
`void __itt_sync_acquired( void *addr)`	スピンループの正常終了を定義します（同期オブジェクトの取得）。
`void __itt_sync_releasing( void *addr)`	同期オブジェクトを解放するコードを開始します。このプリミティブは、ロック解除呼び出しの前に呼び出されます。

各 API は単一の引数 addr を持ちます。このアドレスは、2 つ以上の異なるカスタム同期オブジェクトを区別し

ます。インテル® VTune™ プロファイラーは、アドレスごとに個別のカスタム・オプションを追跡できます。そのため、同じカスタム・オブジェクトでコードの異なる領域へのアクセスを保護するには、それぞれオブジェクトを操作する各 API 呼び出しで、同じ `addr` パラメーターを使用します。

コードに正しく埋め込まれていると、プリミティブはコードが行う同期タイプをインテル® VTune™ プロファイラーに通知します。各 prepare プリミティブは、cancel プリミティブまたは acquired プリミティブとペアにする必要があります。

同期構造には、任意の数の同期オブジェクトを含めることができます。各同期オブジェクトは、ユーザー定義同期 API がオブジェクトの追跡に使用する固有のメモリーハンドルから起動する必要があります。オブジェクトが一意のメモリーポインターを使用する限り、ユーザー定義同期 API を使用して、同時に複数の同期オブジェクトを追跡できます。この動作は、Windows* API の `WaitForMultipleObjects` 関数のオブジェクトのモデル化に似ています。

複数の同期オブジェクトを組み合わせることで、より複雑な同期構造を作成できます。ただし、異なるユーザー定義の同期構造を混在させることは避けてください。これは誤動作の原因となる可能性があります。

API 使用のヒント

ユーザー定義同期 API は、コード内にプリミティブを適切に配置する必要があります。次のガイドに従ってください:

- 同期オブジェクトへのアクセスを取得するコードの直前に prepare プリミティブを配置します。
- コードが同期オブジェクトを待機しなくなった直後に、cancel または acquired プリミティブを配置します。
- コードが同期オブジェクトを保持していないことを示すには、release プリミティブを直前に使用してください。
- 複数の prepare プリミティブを使用して複数のオブジェクトを待機する構造をシミュレートする場合、結果は、オブジェクト・グループ内のオブジェクトに対し最後に呼び出された cancel または acquired プリミティブによって決定されます。

重要な考慮事項とパフォーマンスへの影響:

- prepare プリミティブと acquired プリミティブ間の時間は、影響時間であると見なすことができます。
- プロセッサーがブロックしなくても、prepare プリミティブと cancel プリミティブ間の時間はブロック時間と見なされます。
- ユーザー定義同期 API の使い方を誤ると、統計データが不正確になります。

使用例: ユーザー定義スピン待機

prepare API は、現在のスレッドがメモリー位置でシグナルの待機を開始することをインテル® VTune™ プロ

ファイラーに通知します。この呼び出しは、ユーザー同期構造を呼び出す前に行わなければなりません。prepare API は、acquired または cancel API 呼び出しとペアにする必要があります。
この例は、ユーザー定義のスピン待機構造と組み合わせて使用される prepare および acquired API の使い方を示します:

```
long spin = 1;


__itt_sync_prepare((void *) &spin );
while(ResourceBusy);
// スピン待機
__itt_sync_acquired((void *) &spin );
```

現在のスレッドがユーザー同期構造をテストし、別のスレッドからのシグナルを待機する代わりに、ほかのワークを実行するシナリオでは、キャンセル API を使用すると良いでしょう。この例を示します:

```
long spin = 1;


__itt_sync_prepare((void *) &spin );
while(ResourceBusy)
{
    __itt_sync_cancel((void *) &spin );
     //
     // 有用なワーク
     //
    //  ...
    //
     //  有用なワークが完了したら、この構造体はロック変数をテストし、
     //  再度ロックを取得しようとします。これを実行するには、
     //  事前に prepare API を呼び出す必要があります。
     //
   __itt_sync_prepare((void *) &spin );
}
__itt_sync_acquired((void *) &spin);
```

ロックを取得した後、スレッドがロックを解放する前に releasing API を呼び出す必要があります。次の例は、releasing API の使用方法を示します:

```
long spin = 1;


__itt_sync_releasing((void *) &spin );
// このコードはリソースを解放する必要があります
```

使用例: ユーザー定義同期クリティカル・セクション

この例は、ユーザー定義同期 API を使用して追跡可能なクリティカル・セクションを作成する方法を示します:

```
CSEnter()
{
  __itt_sync_prepare((void*) &cs);
  while(LockIsUsed)
  {
    if(LockIsFree)
    {
     // ロックを実際に取得するコードはここに記述します
    __itt_sync_acquired((void*) &cs);
    }
    if(timeout)
    {
    __itt_sync_cancel((void*) &cs );
    }
  }
}
CSLeave()
{
if(LockIsMine)
    {
        __itt_sync_releasing((void*) &cs);
          // ロックを実際に解除するコードはここに記述します
    }
}
```

このクリティカル・セクションの例は、ユーザー定義の同期プリミティブの使用方法を示しています。次の点に注意してください:

- 各 prepare プリミティブは、acquired プリミティブまたは cancel プリミティブとペアにします。
- prepare プリミティブは、ユーザーコードがユーザーロックを待機する直前に配置します。
- acquired プリミティブは、ユーザーコードがユーザーロックを取得した直後に配置します。
- releasing プリミティブは、ユーザーコードがユーザーロックを解放する直前に配置します。インテル® VTune™ プロファイラーは、スレッドがロックを解放したことを認識する前、他のスレッドが取得したプリミティブを呼び出さないようにします。

使用例: ユーザーレベルの同期バリア

同期 API を使用して、バリアなどの高レベルな構造体をモデル化できます。この例は、同期 API を使用して追跡

可能なバリア構造体を作成する方法を示します:

```
Barrier()
{
    teamflag = false;
    __itt_sync_releasing((void *) &counter);
    InterlockedIncrement(&counter);  // アトミック・インクリメント・プリミティブを使用

    if( counter == thread count )
    {
        __itt_sync_acquired((void *) &counter);
        __itt_sync_releasing((void *) &teamflag);
        teamflag = true;
        counter = 0;
    }
    else
    {
        __ itt_sync_prepare((void *) &teamflag);
        // teamflag を待機
        __ itt_sync_acquired((void *) &teamflag);
    }
}
```

次の点に注意してください:

- このバリアコードには 2 つの同期オブジェクトがあります。`counter` オブジェクトは、スレッドがバリアに入ったことを、示すため、すべてのスレッドから最後のスレッドへギャザーなどのシグナルを通知する際に使用されます。最後のスレッドはバリアに到達すると、`teamflag` オブジェクトを使用してすべてのスレッドに継続が可能であることを通知します。
- 各スレッドがバリアに入ると、`counter` をインクリメントして最後のスレッドに通知することをインテル® VTune™ プロファイラーに伝えるため `__itt_sync_releasing()` を呼び出します
- バリアに入る最後のスレッドは `__itt_sync_acquired()` を呼び出して、他のすべてのスレッドから正常にシグナルを受け取ったことをインテル® VTune™ プロファイラーに通知します。
- バリアに入る最後のスレッドは、`__itt_sync_releasing()` を呼び出して `teamflag` を設定することで、他のすべてのスレッドにバリアの完了を通知することをインテル® VTune™ プロファイラーに通知します。
- 最後のスレッドを除くほかのスレッドは、`__itt_sync_prepare()` プリミティブを呼び出して、最後のスレッドからの `teamflag` シグナルを待機していることをインテル® VTune™ プロファイラーに伝えます。
- 最後に、バリアを出る前に、各スレッドは `__itt_sync_acquired()` プリミティブを呼び出してバリア終了シグナルを正常に受信したことをインテル® VTune™ プロファイラーに通知します。

ジャストインタイム（JIT）API

ジャストインタイム（JIT）プロファイル API を使用して、パフォーマンス・ツールがジャストインタイムで生成されたコードの情報を収集できるようにします。これを行うには、JIT コンパイル済みのコード実行する前に、情報を報告するため この JIT プロファイル API 呼び出しをコード・ジェネレーターに挿入する必要があります。この情報は実行時に収集され、インテル® VTune™ プロファイラーなどのツールが JIT コンパイルコードに関連するパフォーマンス・メトリックを表示するために使用されます。

JIT プロファイル API を使用すると、次のようなシナリオをプロファイルできます:

- JavaScript コードトレースの動的 JIT コンパイル
- OpenCL* アプリケーションにおける JIT 実行
- Java*/.NET* マネージド実行環境
- カスタム ISV JIT エンジン

JIT プロファイル API を使用して、JavaScript* コードのトレースの動的 JIT コンパイル、OpenCL* アプリケーションでの JIT 実行、Java*/.NET マネージド実行環境、およびカスタム ISV JIT エンジンなどの環境をプロファイルできます。

JIT エンジンは実行時にコードを生成し、静的部分を介してプロファイラー・オブジェクト（コレクター）と通信します。実行時に JIT エンジンはプロファイラー・オブジェクトによってトレースファイルに格納された JIT コンパイル済みコードの情報をレポートします。収集が完了すると、プロファイル・ツールは生成されたトレースファイルを参照して JIT コンパイルされたコードを解決します。

JIT プロファイル API を使用して以下を行います:

- トレースベースおよびメソッドベースの JIT コンパイル済みコードのプロファイル
- 関数分割を解析
- インライン関数を調査

JIT プロファイル API の環境変数

JIT プロファイル API には、2 つの環境変数があります:

- `INTEL_JIT_PROFILER32`
- `INTEL_JIT_PROFILER64`

これらの変数には、それぞれ特定のランタイム・ライブラリーへのパスが含まれています。

これらの変数は、JIT API のスタブ実装を JIT API コレクターに置き換えることを通知するために使用されます。JIT API を使用してコードにインストルメント機能を追加し、JIT API スタブ（`libjitprofiling.lib/libjitprofiling.a`）にリンクした後、環境変数が設定されている場合、コードはこれ

らの変数で定義されているライブラリーをロードします。

`ittnotify_collector` がデータを収集できるように、これらの環境変数は必ず設定してください:

Windows*:

```
INTEL_JIT_PROFILER32=<install-dir>¥bin32¥runtime¥ittnotify_collector.dll
INTEL_JIT_PROFILER64=<install-dir>¥bin64¥runtime¥ittnotify_collector.dll
```

Linux*:

```
INTEL_JIT_PROFILER32=<install-dir>/lib32/runtime/libittnotify_collector.so
INTEL_JIT_PROFILER64=<install-dir>/lib64/runtime/libittnotify_collector.so
```

FreeBSD*:

```
INTEL_JIT_PROFILER64=<target-package>/lib64/runtime/libittnotify_collector.so
```

トレースベースおよびメソッドベースの JIT コンパイル済みコードのプロファイル

これは、JIT プロファイル API を使用して、トレースベースおよびメソッドベースの JIT コンパイル済みコードをプロファイルする最も一般的なシナリオです。

```
#include <jitprofiling.h>

if (iJIT_IsProfilingActive() != iJIT_SAMPLING_ON) {
    return;
}

iJIT_Method_Load jmethod = {0};
jmethod.method_id = iJIT_GetNewMethodID();
jmethod.method_name = "method_name";
jmethod.class_file_name = "class_name";
jmethod.source_file_name = "source_file_name";
jmethod.method_load_address = code_addr;
jmethod.method_size = code_size;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&jmethod);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_SHUTDOWN, NULL);
```

使用法

- いずれかの `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` イベントによって、既に報告されているメソッドが上書きされた場合、そのメソッドは無効となります。当該メソッドのメモリー領域は、アンロードされたものとして扱われます。
- 提供された行番号情報に、同じアセンブリー命令（コード位置）に対応する複数のソース行が含まれてい

る場合、プロファイラー・ツールは最初の行番号を適用します。

- 動的に生成されたコードをモジュール名に関連付けることができます。これには `iJIT_Method_Load_V2` 構造を使用します。
- 関数を同じメソッド ID で複数回登録する際に異なるモジュール名を指定すると、プロファイル・ツールは最初に登録されたモジュール名を選択します。異なる JIT エンジン間で同じ関数を区別したい場合、関数ごとに異なるメソッド ID を指定します。ソースファイルなどその他のシンボル情報は、同一である可能性があります。

関数分割を解析

JIT プロファイル API を使用すると、分割された関数を分析できます。このような状況は、リソースが限られた環境で、同じ機能のコードが別々のセグメントで生成、または更新される場合に発生します。状況によっては、このコード生成は重複するライフタイムを持つ時に発生することがあります。

```
#include <jitprofiling.h>

unsigned int method_id = iJIT_GetNewMethodID();

iJIT_Method_Load a = {0};
a.method_id = method_id;
a.method_load_address = 0x100;
a.method_size = 0x20;

iJIT_Method_Load b = {0};
b.method_id = method_id;
b.method_load_address = 0x200;
b.method_size = 0x30;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&a);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&b)
```

使用法

- `iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` イベントによって、既にレポート済みのメソッドが上書きされた場合、そのメソッドは無効となり、そのメモリー領域はアンロードされたものとして扱われます。
- 同じメソッド ID でレポートされるすべてのコード領域は、同一のメソッドに属していると見なされます。シンボル情報（メソッド名、ソースファイル名）は、最初の通知から取得されます。同じメソッド ID を持つ以降のすべての通知は、行番号テーブルの情報のみに基づいて処理されます。
- 異なるソースファイル名と同じメソッド ID で 2 つ目のコード領域を登録した場合、この情報は保存されますが、最初のコード領域の拡張とはみなされません。しかし、このプロファイル・ツールは最初のコード領域のソースファイルを使用するため、パフォーマンス・メトリックを誤ってマッピングする可能性があります。

- 最初のコード領域で使用したソースファイルと同じソースファイルを使用して 2 つ目のコード領域を登録し、同じメソッド ID を使用した場合、ソースファイル自体は破棄されますが、プロファイル・ツールはメトリックをソースファイルに正しくマッピングします。
- 2 番目のコード領域を NULL ソースファイルと同じメソッド ID で登録すると、行番号情報は最初のコード領域のソースファイルに関連付けられます。

インライン関数を調査

JIT プロファイル API を使用して、インライン関数、さらにはネストされたインラインメソッドの多層的な階層構造を調べることができ、パフォーマンスの分布を確認できます。

```
#include <jitprofiling.h>

//                                method_id   parent_id
//   [-- c --]                       3000        2000
//               [---- d -----]      2001        1000
//  [---- b ----]                    2000        1000
// [------------ a ----------------]   1000         n/a

iJIT_Method_Load a = {0};
a.method_id = 1000;

iJIT_Method_Inline_Load b = {0};
b.method_id = 2000;
b.parent_method_id = 1000;

iJIT_Method_Inline_Load c = {0};
c.method_id = 3000;
c.parent_method_id = 2000;

iJIT_Method_Inline_Load d = {0};
d.method_id = 2001;
d.parent_method_id = 1000;

iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED, (void*)&a);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&b);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&c);
iJIT_NotifyEvent(iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED, (void*)&d);
```

使用法

- それぞれのインライン（`iJIT_Method_Inline_Load`）メソッドは、2 つのメソッド ID と関連付ける

必要があります。それは、1 つは直結する親関数に、1 つは自身に関連します。
- 同じ親メソッドのインラインメソッドのアドレス領域は、互いに重複できません。
- 親メソッドとインラインメソッドがレポートされるまで、親メソッドの実行は開始できません。
- 入れ子になったインラインメソッドを呼び出した後は、`iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED` イベントの順番は重要ではありません。
- インラインメソッドまたは上位の親メソッドがオーバーライトされると、インラインメソッドを含む親は無効化され、そのメモリー領域はアンロードされたものとして扱われます。

関連資料

- JIT API を使用したコンパイルとリンク
- JIT API リファレンス

JIT API を使用したコンパイルとリンク

JIT プロファイルを実行するには、次の操作を行います:

- ソースツリーに次のファイルをインクルードします:
  1. コード内で、`<ittapi_dir>¥include` ディレクトリーにある `jitprofiling.h` ファイルを使用します。このヘッダーファイルは、すべての API 関数のプロトタイプとタイプ定義を含みます。
  2. `Ittnotify_config.h`、および、`ittnotify_types.h`、および `jitprofiling.c` は、`<ittapi_dir>/src/ittnotify` ディレクトリーにあります。
- jitprofiling 静的ライブラリーをリンクします:
  1. コード内で、`<ittapi_dir>¥include` ディレクトリーにある `jitprofiling.h` ファイルを使用します。このヘッダーファイルは、すべての API 関数のプロトタイプとタイプ定義を含みます。
  2. `<ittapi_dir>¥build_<target_platform>¥<target_bits>¥bin` ディレクトリーにある、`jitprofiling.lib`（Windows*）または `jitprofiling.a`（Linux*）にリンクします。

使用するプリミティブ	説明
`int iJIT_NotifyEvent( iJIT_JVM_EVENT event_type, void *EventSpecificData );`	この API を使用して、`EventSpecificData` が指すデータとともに `event_type` をエージェントに送信します。レポートされた情報は、任意のプロファイル・ツールのコレクターから取得されたサンプルを関連付けるのに使用されます。
`unsigned int iJIT_GetNewMethodID( void );`	新しいメソッド ID を生成します。この関数を使用して、プロファイラーにレポートされたメソッドに一意で有効なメソッド ID を割り当てる必要があります。この API は新しい一意なメソッド ID を返します。一意なメソッドが利用できない場合、API 関数は 0（ゼロ）を返します。
`iJIT_IsProfilingActiveFlags`	`iJIT_IsProfilingActiveFlags` 列挙子を使用してプロ

使用するプリミティブ	説明
`iJIT_IsProfilingActive( void );`	ファイラーの現在のモード（オフまたはサンプリング）を返します。この API は、デフォルトで `iJIT_SAMPLING_ON` を返し、サンプリングが実行中であることを示します。プロファイラーが実行されていない場合は、`iJIT_NOTHING_RUNNING` を返します。

割り当てられたデータの有効性

イベント固有のデータ（構造体）とエージェント（コレクター）にイベント通知を送信します。構造体内のポインターは、アプリケーションが割り当てたメモリーを指しています。割り当てられたメモリーを解放する責任は開発者にあります。`iJIT_NotifyEvent` メソッドは、これらのポインターを使用してデータをトレースファイルにコピーします。これらのポインターは、`iJIT_NotifyEvent` メソッドが戻った後は使用されません。

JIT API リファレンス

- iJIT_NotifyEvent
- iJIT_IsProfilingActive
- iJIT_GetNewMethodID

iJIT_NotifyEvent

JIT コンパイル済みコードに関する情報をエージェントに報告します。

構文

```
int iJIT_NotifyEvent(iJIT_JVM_EVENT event_type, void EventSpecificData);
```

説明

`iJIT_NotifyEvent` 関数は、`EventSpecificData` が示すデータと `event_type` の通知をエージェントに送信します。レポートされた情報は、任意のプロファイル・ツールのコレクターから取得されたサンプルを関連付けるのに使用されます。この API は、JIT コンパイル後に、JIT コンパイル済みコードの最初のエントリー以前に呼び出す必要があります。

入力パラメーター

パラメーター	説明
`iJIT_JVM_EVENT event_type`	エージェントに送信された通知コード。イベントタイプの完全なリストは以下を参照してください。イベントタイプの完全なリス

パラメーター	説明
	トは以下を参照してください。
`void *EventSpecificData`	イベント固有のデータへのポインター。

`event_type` には次の値を指定できます。

値	説明
`iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED`	JIT メソッドがメモリーにロードされ（JIT コンパイルされた後）、コードが実行される前にこの通知を送信します。`EventSpecificData` には、`iJIT_Method_Load` 構造を使用します。関数 `iJIT_NotifyEvent` の戻り値は不定です。
`iJVM_EVENT_TYPE_SHUTDOWN`	終了するプロファイルにこの通知を送信します。`EventSpecificData` には NULL を使用してください。`iJIT_NotifyEvent` は成功した場合に 1 を返します。
`JVM_EVENT_TYPE_METHOD_UPDATE`	以前報告された動的コードに新しいコンテンツを提供するため、この通知を送信してください。以前のコンテンツは通知後無効になります。次のフィールドで `EventSpecificData` の `iJIT_Method_Load` 構造を使用します。
`JVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED`	インライン動的コードが JIT コンパイルされてメモリーにロードされる場合、親コードが実行を開始する前にこの通知を送信します。`EventSpecificData` には、`iJIT_Method_Inline_Load` 構造を使用します。
`iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED_V2`	JIT された動的コードがメモリーにロードされ、コードが実行される前にこの通知を送信します。`EventSpecificData` には、`iJIT_Method_Load_V2` 構造を使用します。

`EventSpecificData` には、以下の構造を使用できます:

iJIT_Method_Inline_Load 構造体

`iJIT_Method_Inline_Load` 構造を使用して JIT コンパイル済みメソッドを記述する場合、`iJVM_EVENT_TYPE_METHOD_INLINE_LOAD_FINISHED` をイベントタイプとして使用してレポートします。`iJIT_Method_Inline_Load` 構造には次のフィールドが含まれます:

フィールド	説明
`unsigned int method_id`	一意なメソッド ID。このメソッド ID は 999 以下にはできません。`iJIT_GetNewMethodID` API 関数を使用して、有効なメソッド ID を取得するか、ID の一意性と範囲を独自に管理することもできます。
`unsigned int parent_method_id`	固有の直接の親メソッド ID。このメソッド ID は 999 以下にはできません。`iJIT_GetNewMethodID` API 関数を使用して、有効なメソッド ID を取得するか、ID の一意性と範囲を独自に管理することもできます。
`char *method_name`	メソッドの名前を指定します。オプションでクラス名をプレフィックスとして完全なシグネチャーを追加できます。この引数は NULL にできません。
`void *method_load_address`	メソッドコードのベースアドレスを指定します。JIT メソッドでない場合 NULL になることがあります。
`unsigned int method_size`	メソッドがインライン展開されている仮想アドレス。NULL の場合、イベントのデータは受け入れられません。
`unsigned int line_number_size`	行番号テーブルのエントリー数。0 はエントリーなしを示します。
`pLineNumberInfo line_number_table`	行番号情報配列へのポインター。`line_number_size` が 0 である場合 NULL になる可能性があります。行番号情報配列の各エントリーの説明については、`LineNumberInfo` 構造を参照してください。
`char *class_file_name`	クラス名。NULL にできます。
`char *source_file_name`	ソースファイル名。NULL にできます。

iJIT_Method_Load 構造体

`iJIT_Method_Load` 構造を使用して JIT コンパイル済みメソッドを記述する場合、`iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED` をイベントタイプとして使用してレポートします。

`iJIT_Method_Load` 構造には次のフィールドが含まれます:

フィールド	説明
`unsigned int method_id`	一意なメソッド ID。メソッド ID は 999 以下にはできません。`iJIT_GetNewMethodID` API 関数を使用して、有効なメソッド ID を取得するか、独自の ID で一意性と適切な範囲を管理します。

フィールド	説明
`char *method_name`	メソッドの名前を指定します。オプションでクラス名をプレフィックスとして完全なシグネチャーを追加できます。この引数は NULL にできません。
`void *method_load_address`	メソッドコードのベースアドレスを指定します。JIT メソッドでない場合 NULL になることがあります。
`unsigned int method_size`	メソッドがインライン展開されている仮想アドレス。NULL の場合、イベントのデータは受け入れられません。
`unsigned int line_number_size`	行番号テーブルのエントリー数。0 はエントリーなしを示します。
`pLineNumberInfo line_number_table`	行番号情報配列へのポインター。`line_number_size` が 0 である場合 NULL になる可能性があります。行番号情報配列の各エントリーの説明については、`LineNumberInfo` 構造を参照してください。
`unsigned int class_id`	このフィールドは現在使用されません。
`char *class_file_name`	クラス名。NULL にできます。
`char *source_file_name`	ソースファイル名。NULL にできます。
`void *user_data`	このフィールドは現在使用されません。
`unsigned int user_data_size`	このフィールドは現在使用されません。
`iJDEnvironmentType env`	このフィールドは現在使用されません。

iJIT_Method_Load_V2 構造体

`iJIT_Method_Load_V2` 構造を使用して JIT コンパイル済みメソッドを記述する場合、`iJVM_EVENT_TYPE_METHOD_LOAD_FINISHED_V2` をイベントタイプとして使用してレポートします。

`iJIT_Method_Load_V2` 構造には次のフィールドが含まれます:

フィールド	説明
`unsigned int method_id`	一意なメソッド ID。メソッド ID は 999 以下にはできません。`iJIT_GetNewMethodID` API 関数を使用して、有効なメソッド ID

フィールド	説明
	を取得するか、独自の ID で一意性と適切な範囲を管理します。
`char *method_name`	メソッドの名前を指定します。オプションでクラス名をプレフィックスとして完全なシグネチャーを追加できます。この引数は NULL にできません。
`void *method_load_address`	メソッドコードのベースアドレスを指定します。JIT メソッドでない場合 NULL になることがあります。
`unsigned int method_size`	メソッドがインライン展開されている仮想アドレス。NULL の場合、イベントのデータは受け入れられません。
`unsigned int line_number_size`	行番号テーブルのエントリー数。0 はエントリーなしを示します。
`pLineNumberInfo line_number_table`	行番号情報配列へのポインター。`line_number_size` が 0 である場合 NULL になる可能性があります。行番号情報配列の各エントリーの説明については、`LineNumberInfo` 構造を参照してください。
`char *class_file_name`	クラス名。NULL にできます。
`char *source_file_name`	ソースファイル名。NULL にできます。
`char *module_name`	モジュール名。NULL にできます。モジュール名は異なる JIT エンジンを区別するのに役立ちます。

LineNumberInfo 構造

`LineNumberInfo` 構造を使用して、コード領域の行番号情報に単一のエントリーを作成します。行番号エントリーの表は、レポートされたコード領域のソースコードへのマップを示す情報を提供します。インテル® VTune™ プロファイラーは、行番号情報を使用してサンプル（仮想アドレス）を行番号に対応付けます。同じソース行に対し異なるコードアドレスがレポートされることがあります:

オフセット	行番号
1	2
12	4

オフセット	行番号
15	2
18	1
21	30

インテル® VTune™ プロファイラーは、クライアント・データを使用して次の表を構成します:

コードのサブレンジ	行番号
0-1	2
1-12	4
12-15	2
15-18	1
18-21	30

`LineNumberInfo` 構造には次のフィールドが含まれます。

フィールド	説明
`unsigned int Offset`	メソッドの先頭からの Opcode バイトオフセット。
`unsigned int LineNumber`	一致するソース行番号のオフセット（ソースファイルの先頭から）

戻り値

戻り値は、`iJIT_JVM_EVENT` に依存します。

iJIT_IsProfilingActive

エージェントの現在のモードを返します。

構文

```
iJIT_IsProfilingActiveFlags JITAPI iJIT_IsProfilingActive(void);
```

説明

関数 `iJIT_IsProfilingActive` は、エージェントの現在のモードを返します。

入力パラメーター

なし

戻り値

`iJIT_SAMPLING_ON` はエージェントが実行中であることを示します。また、`iJIT_NOTHING_RUNNING` はエージェントが実行されていないことを示します。

iJIT GetNewMethodID

新しい一意のメソッド ID を生成します。

構文

```
unsigned int iJIT_GetNewMethodID(void);
```

説明

`iJIT_GetNewMethodID` 関数は、呼び出されるたびに新しいメソッド ID を生成します。一意のメソッド ID を生成する独自のメカニズムがない場合、この API を使用して、エージェントに報告されたメソッドまたはトレースの一意で有効なメソッド ID を取得します。

入力パラメーター

なし

戻り値

新しい一意のメソッド ID をします。一意なメソッドが利用できない場合、API 関数は 0（ゼロ）を返します。

ITT API リファレンス・コレクター

これは、ITT API の動的部分のリファレンス実装であり、ITT API 関数呼び出しのトレースデータをログファイルに記録します。

このソリューションを使用するには、コレクターを共有ライブラリーとしてビルドし、ライブラリーへの完全なパスを `INTEL_LIBITTNOTIFY64` または `INTEL_LIBITTNOTIFY32` 環境変数に指定します:

Linux*

```
make
export INTEL_LIBITTNOTIFY64=<build_dir>/libittnotify_refcol.so
```

FreeBSD*

```
make
setenv INTEL_LIBITTNOTIFY64 <build_dir>/libittnotify_refcol.so
```

デフォルトでは、ログファイルは Temp ディレクトリーに保存されます。場所を変更するには、`INTEL_LIBITTNOTIFY_LOG_DIR` 環境変数を使用します:

Linux*

```
export INTEL_LIBITTNOTIFY_LOG_DIR=<log_dir>
```

FreeBSD*

```
setenv INTEL_LIBITTNOTIFY_LOG_DIR <log_dir>
```

この実装では、ITT API 関数呼び出しの一部にログ記録機能を追加しています。その他の ITT API 関数呼び出しにログ記録を追加することも歓迎します。このソリューションでは、ログ出力に `printf` 形式を使用する、異なるログレベルを持つ 4 つの関数を提供します:

```
LOG_FUNC_CALL_INFO(const char *msg_format, ...);
LOG_FUNC_CALL_WARN(const char *msg_format, ...);
LOG_FUNC_CALL_ERROR(const char *msg_format, ...);
LOG_FUNC_CALL_FATAL(const char *msg_format, ...);
```